Chapter EIGHTEEN
Parallel Streams

Exam Objectives

Use parallel Streams including reduction, decomposition, merging processes, pipelines and performance.

What is a Parallel Stream?

Until now, all the examples of this section have used sequential streams, where each element are processed one by one.

In contrast, parallel streams split the stream into multiple parts. Each part is processed by a different thread at the same time (in parallel).

Under the hood, parallel streams use the Fork/Join Framework (which we'll review in a later chapter).

This means that by default, the number of threads available to process parallel streams equals the number of available cores of the processor of your machine.

The advantage of using parallel streams over the Fork/Join Framework is that they are easier to use.

To create a parallel stream just use the `parallel()` method:

```
Stream<String> parallelStream =
                  Stream.of("a","b","c").parallel();
```

To create a parallel stream from a `Collection` use the `parallelStream()` method:

```
List<String> list = Arrays.asList("a","b","c");
Stream<String> parStream = list.parallelStream();
```

How Parallel Streams work?

Let's start with the simplest example:

```
Stream.of("a","b","c","d","e")
    .forEach(System.out::print);
```

Printing a list of elements with a sequential stream will output the expected result:

```
abcde
```

However, when using a parallel stream:

```
Stream.of("a","b","c","d","e")
    .parallel()
    .forEach(System.out::print);
```

The output can be different for each execution:

```
cbade // One execution
cebad // Another execution
cbdea // Yet another execution
```

The reason is the decomposing of the stream on parts and their processing by different threads we talked about before.

So parallel streams are more appropriate for operations where the order of processing doesn't matter and that don't need to keep a state (they are stateless and independent).

An example to see this difference is the use of `findFirst()` vs. `findAny()`.

In a previous chapter, we mentioned that `findFirst()` method returns the first element of a stream. But when using parallel streams and this is decomposed in multiple parts, this method has to "know" which element is the first one:

```
long start = System.nanoTime();
String first = Stream.of("a","b","c","d","e")
    .parallel().findFirst().get();
long duration = (System.nanoTime() - start) / 1000000;
System.out.println(
    first + " found in " + duration + " milliseconds");
```

The output:

```
a found in 2.436155 milliseconds
```

Because of that, if the order doesn't matter, it's better to use `findAny()` with parallel streams:

```
long start = System.nanoTime();
String any = Stream.of("a","b","c","d","e")
    .parallel().findAny().get();
long duration = (System.nanoTime() - start) / 1000000;
System.out.println(
    any + " found in " + duration + " milliseconds");
```

The output:

```
c found in 0.063169 milliseconds
```

As a parallel stream is processed, well, in parallel, is reasonable to believe that it will be processed faster than a sequential stream. But as you can see with `findFirst()`, this is not always the case.

Stateful operations, like :

```
Stream<T> distinct()
Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> comparator)
Stream<T> limit(long maxSize)
Stream<T> skip(long n)
```

Incorporate state from previously processed elements and may need to go through the entire stream to produce a result, so they are not a good fit for parallel streams.

By the way, you can turn a parallel stream into a sequential one with the `sequential()` method:

```
stream
   .parallel()
      .filter(..)
         .sequential()
            .forEach(...);
```

Check if a stream is parallel with `isParallel()` :

```
stream.parallel().isParallel(); // true
```

And turn an ordered stream into a unordered one (or ensure that the stream is unordered) with `unordered()` ;

```
stream
   .parallel()
      .unordered()
         .collect(...);
```

But don't believe that by first executing the stateful operations and then turning the stream into a parallel one, the performance will be better in all cases, or worse, the entire operation may run in parallel, like the following example:

```
double start = System.nanoTime();
Stream.of("b","d","a","c","e")
    .sorted()
    .filter(s -> {
        System.out.println("Filter:" + s);
        return !"d".equals(s);
    })
    .parallel()
    .map(s -> {
        System.out.println("Map:" + s);
        return s += s;
    })
    .forEach(System.out::println);
double duration = (System.nanoTime() - start) / 1_000_000;
System.out.println(duration + " milliseconds");
```

One might think that the stream is sorted and filtered sequentially, but the output shows something else:

```
Filter:c
Map:c
cc
Filter:a
Map:a
aa
```

```
Filter:b
Map:b
bb
Filter:d
Filter:e
Map:e
ee
79.470779 milliseconds
```

Compare this with the output of the sequential version (just comment `parallel()`):

```
Filter:a
Map:a
aa
Filter:b
Map:b
bb
Filter:c
Map:c
cc
Filter:d
Filter:e
Map:e
ee
1.554562 milliseconds
```

In this case, the sequential version performed better.

But if we have an independent or stateless operation, where the order doesn't matter, let's say, counting the number of odd numbers in a large range, the parallel version will perform better:

```
double start = System.nanoTime();
long c = IntStream.rangeClosed(0, 1_000_000_000)
            .parallel()
            .filter(i -> i % 2 == 0)
            .count();
double duration = (System.nanoTime() - start) / 1_000_000;
System.out.println("Got " + c + " in " + duration + " milliseconds");
```

The parallel version output:

```
Got 500000001 in 738.678448 milliseconds
```

The sequential version output:

```
Got 500000001 in 1275.271882 milliseconds
```

In summary, parallel streams don't always perform better than sequential streams.

This, the fact that parallel streams process results independently and that the order cannot be guaranteed are the most important things you need to know.

But in practice, how do you know when to use sequential or parallel streams for better performance?

Here are some rules:

- For a small set of data, sequential streams are almost always the best choice due to the overhead of the parallelism.

- When using parallel streams, avoid stateful (like `sorted()`) and order-based (like `findFirst()`) operations.
- Operations that are computationally expensive (considering all the operation in the pipeline), generally have a better performance using a parallel stream.
- When in doubt, check the performance with an appropriate benchmark.

Reducing Parallel Streams

In concurrent environments, assignments are bad.

This is because if you mutate the state of variables (especially if they are shared by more than one thread), you may run into many troubles to avoid invalid states.

Consider this example, which implements the factorial of `10` in a very particular way:

```
class Total {
    public long total = 1;
    public void multiply(long n) { total *= n; }
}
...
Total t = new Total();
LongStream.rangeClosed(1, 10)
    .forEach(t::multiply);
System.out.println(t.total);
```

Here, we are using a variable to gather the result of the factorial. The output of executing this snippet of code is:

```
3628800
```

However, when we turn the stream into a parallel one:

```
LongStream.rangeClosed(1, 10)
    .parallel()
    .forEach(t::multiply);
```

Sometimes we get the correct result and other times we don't.

The problem is caused by the multiple threads accessing the variable `total` concurrently. Yes, we can synchronize the access to this variable (as we'll see in a later chapter), but that kind of defeats the purpose of parallelism (I told you assignments are bad in concurrent environments).

Here's where `reduce()` comes in handy.

If you remember from the previous chapter, `reduce()` combines the elements of a stream into a single one.

With parallel streams, this method creates intermediate values and then combines them, avoiding the "ordering" problem while still allowing streams to be processed in parallel by eliminating the shared stated and keep it inside the reduction process.

The only requirement is that the applied reducing operation must be associative.

This means that the operation `op` must follow this equality:

```
(a op b) op c == a op (b op c)
```

Or:

```
a op b op c op d == (a op b) op (c op d)
```

So we can evaluate `(a op b)` and `(c op d)` in parallel.

Returning to our example, we can implementing it using `parallel()` and `reduce()` in this way:

```
long tot = LongStream.rangeClosed(1, 10)
                .parallel()
                .reduce(1, (a,b) -> a*b);
System.out.println(tot);
```

When we execute this snippet of code, it produces the correct result every time (`3628800`).

And if we time the execution of the first snippet (47.216181 milliseconds) and this last one (3.094717 milliseconds), we can see a great improvement in performance. Of course, these values (and the other ones presented in this chapter) depend on the power of the machine, but you should get similar results.

We can also apply reduction to the example presented at the beginning of this chapter to process the string in parallel while maintaining the order:

```
String s = Stream.of("a","b","c","d","e")
             .parallel()
             .reduce("", (s1, s2) -> s1 + s2);
System.out.println(s);
```

The output:

```
abcde
```

These are simple examples, but reduction is hard to get it right sometimes, so it's best to avoid shared mutable state and use stateless and independent operations to ensure parallel streams produce the best results.

For instance, remember this example from the last chapter?

```
int total = IntStream.of(1, 2, 3, 4, 5, 6)
             .reduce( 4, (sum, n) -> sum + n );
```

In this example, the first parameter is not really an identity since `4+n` is not equal to `n` .

When I made the stream parallel, instead of getting the expect result (`25`), I got `45` :

```
int total = IntStream.of(1, 2, 3, 4, 5, 6)
             .parallel()
             .reduce( 4, (sum, n) -> sum + n );
```

Why?

Because the stream is divided into parts, and the accumulator function is applied to each one independently, which means that `4` is added not only to the first element, but to the first element of each part.

Also, as we also saw in the last chapter, the version of `reduce()` that takes three arguments is particularly useful in parallel streams.

For instance, if we take the example from the last chapter that sums the length of some strings and make it parallel:

```
int length = Stream.of("Parallel", "streams", "are", "great")
               .parallel()
               .reduce(0,
                  (accumInt, str) ->
                     accumInt + str.length(), //accumulator
                  (accumInt1, accumInt2) ->
                     accumInt1 + accumInt2); //combiner
```

We can see what happens to get the result (`23`):

1. The accumulator function is applied to each element in no particular order. For example:

```
0 + 5 // great
0 + 3 // are
0 + 8 // Parallel
0 + 7 // streams
```

2. The results of the accumulator function are combined:

```
5 + 3 // 8
8 + 7 // 15
8 + 15 // 23
```

When we are returning a reduced value of type `U` from elements of type `T` , and in a parallel stream the elements are divided into `N` intermediate results of type `U` , it makes sense to have a function that knows how to combine those values of type `U` into a single result.

Because of that, if we are not using different types, the accumulator function **IS** the same as the combiner function.

Finally, just like `reduce()` , we can safely use `collect()` with parallel streams if we follow the same requirements of associativity and identity, like for example, for any partially accumulated result, combining it with an empty result container must produce an equivalent result.

Or, if we are grouping with the `Collectors` class and ordering is not important, we can use the method `groupingByConcurrent()` , the concurrent version of `groupingBy()` .

Key Points

- Parallel streams split the stream into multiple parts. Each part is processed by a different thread at the same time (in parallel).
- To create a parallel stream from another stream, use the `parallel()` method.
- To create a parallel stream from a `Collection` use the `parallelStream()` method.
- Parallel streams are more appropriate for operations where the order of processing doesn't matter and that don't need to keep a state (they are stateless and independent).
- You can turn a parallel stream into a sequential one with the `sequential()` method.
- You can check if a stream is parallel with `isParallel()` .

- You can turn an ordered stream into a unordered one (or ensure that the stream is unordered) with `unordered()`.
- Parallel streams don't always perform better than sequential streams.
- With parallel streams, `reduce()` creates intermediate values and then combines them, avoiding the "ordering" problem while still allowing streams to be processed in parallel by eliminating shared (mutating) state and keeping it inside the reduction process.
- The only requirement is that the applied reducing operation must be associative: `(a op b) op c == a op (b op c)`.

Self Test

1. Given:

```
public class Question_18_1 {
    public static void main(String[] args) {
        OptionalInt sum = IntStream.rangeClosed(1, 10)
                            .parallel()
                            .unordered()
                            .reduce(Integer::sum);
        System.out.println(sum.orElse(0));
    }
}
```

What is the result?
A. `0`
B. `55` is printed all the time
C. Sometimes `55` is printed
D. An exception occurs at runtime

2. Which of the following statements is true?
A. You can call the method `parallel()` in a `Collection` to create a parallel stream.
B. Operations that are computationally expensive, generally have a better performance using a sequential stream.
C. `filter()` is a stateless method.
D. Parallel streams always perform better than sequential streams.

3. Given:

```
public class Question_18_3 {
    public static void main(String[] args) {
        OptionalDouble avg =
                IntStream.rangeClosed(1, 10)
                            .parallel()
                            .average();
        System.out.println(avg.getAsDouble());
    }
}
```

What is the result?
A. `5.5` is printed all the time
B. Sometimes `5.5` is printed
C. Compilation fails
D. An exception occurs at runtime

4. Given:

```
public class Question_17_4 {
    public static void main(String[] args) {
```

```
        IntStream.of(1, 1, 3, 3, 7, 6, 7)
            .distinct()
            .parallel()
            .map(i -> i*2)
            .sequential()
            .forEach(System.out::print)
    }
}
```

What is the result?
A. It can print `142612` sometimes
B. `1133767`
C. It can print `3131677` sometimes
D. `261412`

Open answers page

Do you like what you read? Would you consider?

Buying the print/kindle version from Amazon

Buying the PDF/EPUB/MOBI versions from Leanpub

Buying the e-book version from iTunes

Buying the e-book version from Kobo

Buying the e-book version from Scribd

Do you have a problem or something to say?

Report an issue with the book

Contact me

---

17. Peeking, Mapping, Reducing and Collecting

19. Exceptions